EXHIBIT 11

Appendix

Verilog HDL Design

This appendix introduces the hardware description language, *Verilog*. It describes how to use Verilog and create your own hardware models. After a short introduction, the design flow and the level of designs using Verilog are explained. Then the basic Verilog syntax is explained with basic examples. We include RTL code and simulation environments, so that users can simulate the rendering processor on their PC or workstation environment. In addition, the synthesis scripts are also included.

A.1 Introduction to Verilog Design

Verilog is a *hardware description language* (HDL), which is a language to describe a digital system such as microprocessor, application-specific unit, memory, or a simple flip–flop. Anyone can describe any hardware at any level of detail by using a hardware description language.

Verilog was developed initially as a proprietary hardware modeling language by Gateway Design Automation Inc. around 1984. It is very similar to traditional computer language such as C. At that time, Verilog was not standardized and the language became modified in almost all the revisions that came out between 1984 and 1990. A Verilog simulator was first used in early 1985 and was extended substantially through to 1987. In 1990, Cadence Design Systems decided to acquire Gateway, so Cadence became the owner of the Verilog language, and continued to market Verilog as both a language and a simulator. At the same time, Synopsys was marketing the top-down design methodology, using Verilog. This was the starting point of powerful digital system design. In 1990, to survive in a tough industry that used VHDL, Cadence decided to make Verilog an "open" language. Cadence organized the Open Verilog International (OVI) in 1991 and, in the meantime, the popularity of Verilog was rising exponentially.

Mobile 3D Graphics SoC: From Algorithm to Chip Jeong-Ho Woo, Ju-Ho Sohn, Byeong-Gyu Nam and Hoi-Jun Yoo
 ISBN: 978-0-470-82377-4

The need for a universal standard was recognized at this time. The directors of OVI asked the IEEE to form a working committee to establish Verilog as an IEEE standard. The working committee 1364 was formed in mid 1993, and finally in 1995 the standard, which combined both the Verilog language syntax and the PLI in a single volume, was passed as IEEE standard 1364(1995).

A.2 Design Level

Verilog allows a digital system to be designed at a wide range of abstractions – at behavior level, at register transfer level (RTL), at gate level, at switch level, and so on. Among these, behavior level, RTL level, and gate level design are frequently used for digital system design.

A.2.1 Behavior Level

The behavior level is used to describe a system intuitively. Therefore, this level is frequently used to verify algorithms or system behavior like a high-level language. This level is not focused on synthesizability or structural realization of the design.

A.2.2 Register Transfer Level

After high-level verification at the behavior level, RTL code is used to specify the characteristics of a circuit or system by operations and the transfer of data between the registers. This level is focused on real implementation of the design, so the design should be carefully described according to synthesizable syntax or semantics. At this level the design contains explicit clocks, so RTL design contains exact timing bounds.

A.2.3 Gate Level

After synthesizing RTL code, gate-level code is generated. Within this level, the characteristics of a system are described by standard cell libraries and their logical links and timing properties. All signals are discrete signals. They can only have definite logical values (0, 1, X, Z). The usable operations are predefined logic primitives (gates AND, OR, NOT etc.). Gate-level code is generated by tools like synthesis tools, and this net list is used for gate-level simulation and for back-end design.

Design flow is one of the most difficult decisions. Like other HDLs, Verilog also allows both bottom-up and top-down design.

- **Bottom-up design** This is a type of traditional design flow. Each design is performed at the gate level using standard gates such as AND, OR, NOR, and NAND.

Designers decide which gates will be used and where those gates will be placed. With increasing design complexity, this approach is nearly impossible to maintain. Emerging large-scale systems consist of ASIC or microprocessors with a complexity of millions of transistors. These traditional bottom-up designs have to give way to new, structural, hierarchical design methods.

- **Top-down design** Designers can concretize their ideas from concept to silicon implementation. The major advantages of top-down design are early testing, easy change between different technologies, and a structured system design. Details of top-down design will be discussed below.

A.3 Design Flow

Figure A.1 shows a typical design flow of a digital system using Verilog. In short, the design flow can be divided into two steps: front-end and back-end. It consists of about nine or ten design steps, which will be described briefly.

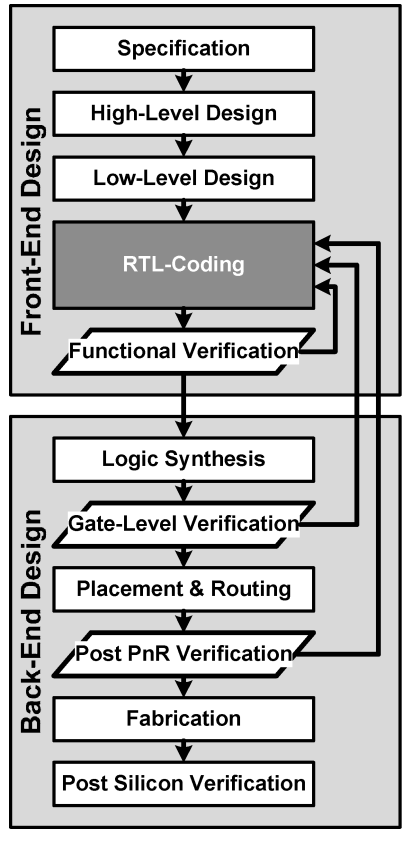


Figure A.1 Typical design flow

A.3.1 Specification

The designer decides what the target performance will be, what the important parameters of the design are, what the interfaces are, and so on (Figure A.2). For example, when designing a microprocessor, the designer should decide the target performance, the reset behavior, the interfaces with other blocks, and input/output format. Normally, the specification does not need a technical tool; designers often use documentation tools such as Microsoft Word, PowerPoint, Visio, and so on.

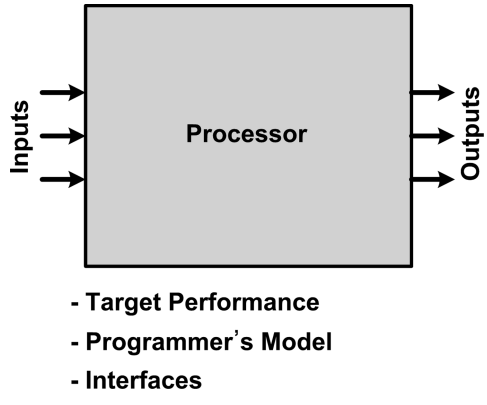


Figure A.2 Specification

A.3.2 High-level Design

After deciding the specification, the designer should describe the target system in high-level language (Figure A.3). Algorithm verification or block definitions are performed in this stage. To design a microprocessor, how to divide functional blocks and how to communicate between them should be decided. For algorithm verification or block definition, a high-level language such as C or C++ or Verilog behavior-level description is widely used in this stage.

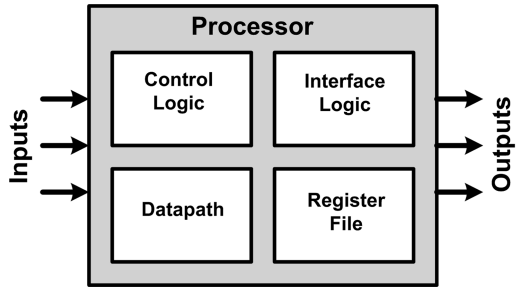


Figure A.3 High-level design

A.3.3 Low-level Design

The designer defines the details about how each block works. In this stage, you should define how to design a finite state machine, registers, and control units of each block (Figure A.4). Of course, the detailed timing diagrams of interfaces are defined with waveforms. It is strongly recommended that you make all items of the design clear in this phase.

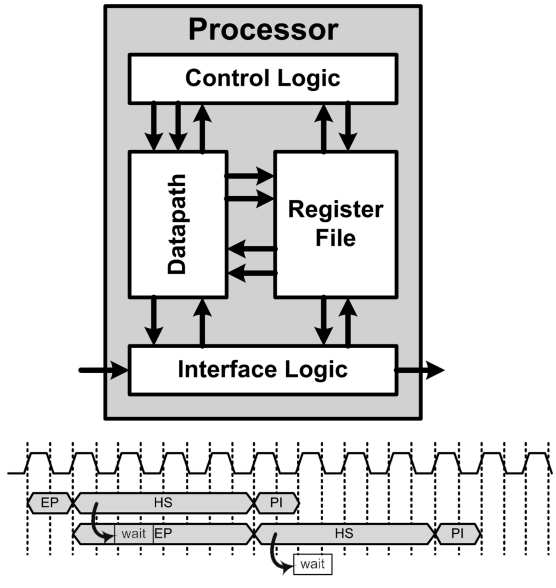


Figure A.4 Low-level design

A.3.4 RTL Coding

In RTL coding, the low-level design is implemented into a hardware description language such as Verilog or VHDL.

```
module processor( clk,      //Clock
                  reset     //Reset
                  inputs;
                  outputs);
      input clk;
      input reset;
      input inputs;
      output      outputs

      controller        proc_ctrl(clk, reset, signals)
      datapath          proc_dp(clk, reset, signals)
      registerfile      proc_rf(clk, reset, signals)
      interfaces        proc_if(clk, reset, signals)
endmodule
```

A.3.5 Simulation

The functionality and timing characteristics of a system should be verified at every step of its design (Figure A.5). A test bench is needed that generates input signals, dummy blocks that are behavioral models of neighboring blocks, and simulators. The initial simulation is done by using RTL codes for functional verification of blocks. After that, gate-level simulation is performed to verify timing bounds as well as functionalities. The final simulation is performed after placement and routing (PnR). After PnR, the design contains wire delays and parasitic resistances and capacitances, so verification is needed that the design meets the target specification.

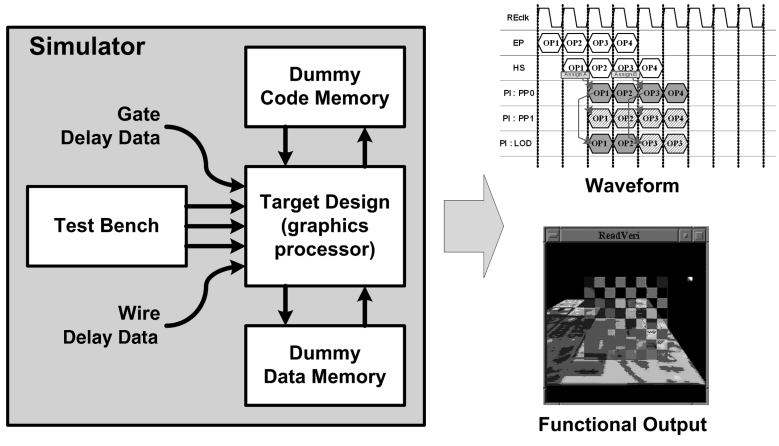


Figure A.5 Simulation

After the simulations, waveform outputs and functional outputs can be obtained, to check functionality and timing bounds.

A.3.6 Synthesis

In the synthesis stage, the RTL code is converted into a gate-level net list (Figure A.6). A synthesis tool such as a design compiler or Synplify takes the RTL code and technology library files as inputs and generates gate-level code that contains standard cells and link data. Basically, the synthesis tool reports on whether or not the timing information meets the timing requirements. However, it does not include wire delays, only gate propagation delays.

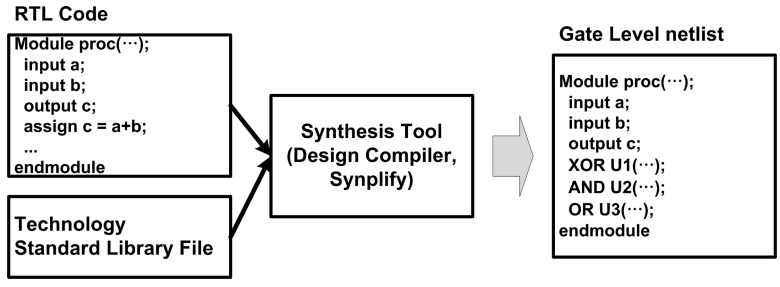


Figure A.6 Synthesis

A.3.7 Placement and Routing

In this stage, the gate-level netlists are converted into a physical layout (Figure A.7). All gates and flip–flops in a gate-level netlist are placed; and clock tree synthesis and global signals such as "reset" are routed. After that, local signals are routed. Then the final GDS file that will be delivered to the foundary, and final net list which includes wire delays as well as gate delays, are generated.

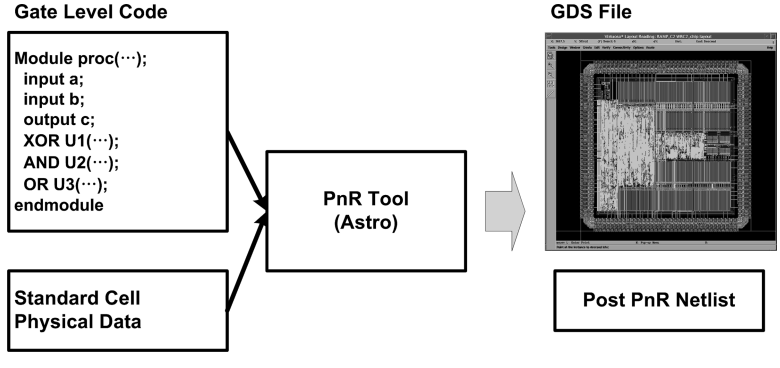


Figure A.7 Placement and routing

A.4 Verilog Syntax

The basic Verilog syntax will be described. The goal of this section is to show how to write Verilog, but it does not cover all the syntax. More information is available at, for example, www.verilog.com and www.verilogtutorial.info.

A.4.1 Modules

The basic unit of Verilog modeling is the *module*. As in the C language, you can use lower-case letters, capital letters, numbers, and underscores (_). Of course, it is case-sensitive. Below we show the basic configuration of a module.

Module Example

```
module  module_name (port-list);     //Declare all inputs and outputs
port declaration
     //Detailed declaration of inputs and outputs, including bitwidth
     //For example:
     //input  clk, res; → 1-bit input signals
     //input  [7:0] bus; → 8-bit input signal
     //output  done; → 1-bit output signal
     //inout  [3:0] dbus; → 4-bit inout signal
     //inout means bidirectional signal
net declaration
     //Declaration of internal net of the block
     //wire  enable;
     //wire [7:0] int_bus;
reg declaration
     //Register declaration
     //reg  enable;
     //reg [3:0] number;
     //You should declare all ports or values which are used in
     //initial, task, function, and always statements
parameter declaration
     //Parameter declaration
     //parameter WORD=32;
     //parameter BW=8;
     //reg [BW:0] temp_out; → example of usage
assign statement
     //Combinational logic declaration
function statement
always statement
Instantiation
     //Instantiate sub-modules
endmodule
     //Declaration of the end of the module
```

Initial Statement Example

"Initial" and "always" statements are *event-driven*. The details of these statements will be discussed later. Here only the configuration is discussed.

```
initial
     //Initial statement does not require an event list.
begin
     //If the initial includes more than one statement, 'begin-end' command
     clk = 0;   //Bind those statements into one command
     reg = 1;   //Use begin-end command with event-driven statements
end
     //Include more than one statement.
```

Always Statement Examples

```
always @(posedge Clk)
      //Every positive edge of Clk signal drives this statement.
always @(posedge Clk or negedge reset)
      //Every positive edge of Clk or negative edge or reset drives.
always @(a or b or c)
      //When one of the conditional signals changes,
      //this statement is executed.
always @(posedge Clk)
begin
      //An always statement also requires 'begin-end'.
      command_latch <= command_in; //Command
      data_latch <= data_in;
end
```

Examples of Bad Expressions

```
always @(Clk)
      //This is a correct expression in terms of grammar.
      //However, the clock signal is edge-triggered.
      //Therefore this statement will incur a timing violation.
always @(a or b or posedge Clk)
      //Do not mix level-triggered and edge-triggered events.
```

A.4.2 Logic Values and Numbers

Verilog supports four logic values:

0	Logic value 0
1	Logic value 1
X	Unknown signal means "don't care"
Z	High-impedance value

To describe logic values or numbers in Verilog, follow the syntax:

```
<bit width>'<base><value>
```

Here, <bit width> declares the width of the data bit in decimal (the default is 32-bit); and <base> declares the base number of the data (default = decimal)

b, B: binary o,O: octal

d,D: decimal h,H: hexadecimal

The <value> declares the value of data.

When the <base> is decimal, you can declare data without <bit width> and <base>, and you can use underscore (_) for convenience.

Data	Bit width	Base	Binary digit
10	32	10	0000. . .01010
1'b1	1	2	1
8'hf1	8	16	11110001
4'bx	4	2	Xxxx
8'b0000_11xx	8	2	000011xx
'hf_f	32	16	0000 011111111
4'd5	4	10	0101

A.4.3 Data Types

To declare signals or variables you have two choices, the "register" type or the "net" type. You should declare the type for all variables.

Register type	Declare using "reg"
	Used for latch or flip–flop
	It stores values until the next event occurs
Net type	Declare using "wire"
	Used for combinational logic or intermediate signals for links

When the data is used on the left side there is no limitation. When the data is used on the right side you should follow two rules:

- Use the "reg" type signals only in always, initial, task, and function statements.
- Use the "wire" type signals only in assign statements.

You can omit the wire declaration only when the "net" type signal is in the port list.

```
//Example 1
module  DFF(Clk, D, Q);
      input  Clk;
      input  D;
      output  Q;
      wire  Clk, D; //You can omit this statement
      reg  Q;
      always @(posedge Clk)  Q <=D;
endmodule

//Example 2
module  Func_1(In1, enable, Q1, Q2);
      input  In1;
      input[2:0]  enable;
      output  Q1;
      output  Q2;
      wire  In1; //You can omit this statement
      wire [2:0] enable //You can omit this statement
      wire  int_sig //You can omit this statement
                          but it is useful for debugging
```

```
    wire  Q1, Q2 //You can omit this statement
    assign  Q1 = enable[2]  enable[0] ;
    nand  nd1(In1, enable[2], int_sig);
    nand  nd2(int_sig, enable[0], Q2);
endmodule
```

When you declare multibit signals, you can declare the bit width of the signal in the format [MSB:LSB]:

```
input [3:0] enable;  //4-bit input signal enable
wire [7:0] bus;  //8-bit net signal bus
output [4:0] out;  //5-bit output signal out
```

Of course you can select bits from multibit signals, or you can assign part of the multibit signal:

```
wire  SB;
wire [2:0]  DB;
assign SB = dbus [7];
assign = dbus [6:4];
assign  dbus[3:0] = enable;
```

You can assign a register array. This is frequently used for a memory block or register file:

```
reg [15:0]  mem[0:127];  //16-bit signal * 128 entries
```

However in this case you *cannot* directly select part of the signal in a register array; you can access the signal only entry level. And Verilog does *not* support multidimensional arrays like C or C++.

```
reg [15:0]   mem[0:127]
//To access the last 4bit of the first entry of the mem array.

//Incorrect example
assign out1 = mem[0][4:0]
//Correct example
wire [15:0]  temp;
assgign temp = mem[0];
assign out1 = temp[4:0];
```

A.4.4 Operators

Verilog supports arithmetic operators, bitwise operators, bit-reduction operators, logic operators, relative operators, shift operators, conditional operators, and concatenation. The operators are listed below.

```
wire a, b, c;
wire [4:0] d, e;
```

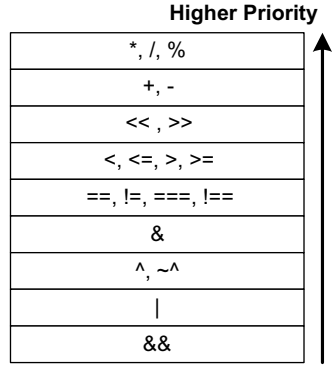
Operator		Meaning
arithmetic	+	Add
	-	Sub
	*	Multiplication
	/	Division
	%	Modulo operation
Bit	~	NOT
	&	AND
	|	OR
	^	XOR
	~^	XNOR
Bit Reduction	&	AND
	~&	NAND
	|	OR
	~|	NOR
	^	XOR
	~^	XNOR

Operator		Meaning
Logic Operation	!	Logical NOT
	&&	Logical AND
	||	Logical OR
	==	Equal
	!=	Not equal
	===	Equal including (x, z)
	!==	Not equal including (x, z)
Relative	<	Less than
	<=	Less than or equal to
	>	Larger than
	>=	Larger than or equal to
Shift	<<	Left Shift
	>>	Right Shift
ETC	? :	Conditional
	{ }	Concatenation

```
//Arithmetic operations
assign  c = a+b;
assign  c = a-b;
assign  c = a*b;
assign  c = a/b;
assign  c = a%b;
      //If the process library supports these arithmetic operations, the
        operators are
      //simply converted into an arithmetic unit, called design ware.
      //The types of those arithmetic units are decided
      //according to timing constraints.
      //If the process library does not support design ware,
      //these expressions are not synthesizable.
//Bit operations
assign  c = ~a;
assign  c = a&b;
//Bit-reduction operations
assign  c = &d;
      //c = d[4] & d[3] & d[2] & d[1] & d[0];
assign  c = |d;
      //c = d[4] | d[3] | d[2] | d[1] | d[0];
//Logical operations
assign  c = (a == 1) && (b==1);
assign  c = (a >= 1) || (b !=3);
      //When you want to decide a certain signal with logical conditions,
      //use logical operators.
      //Since === and !== are not synthesizable, use them carefully.
```

```
//Shift operations
assign  d = a << 3;
      //Shift three bits to the left
assign  e = d >>2;
      //shift two bits to the right
//Conditional operation
assign  d = (a == 0)? 11000 : 00001;
      //If the condition is true, the procedure is selected.
      //If not, the latter is selected.
//Concatenations
assign  {a,b,c} = d[2:0];
      //Assign the last three bits of d to a, b, c, respectively
assign  d = {a, b, c, 2'b11};
```

The *priority of the operators* is very similar that of the C and C++ languages:



When you use concatenation, you also can use repeated expressions:

```
wire [7:0]  temp;
assign temp = {4{2'b10}};  //temp = 8'b10101010
wire [15:0]  word;
wire [31:0]  double
assign double = {{16{word[15]}}, word};
```

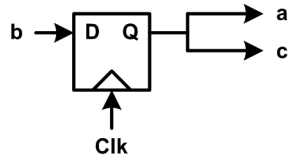
A.4.5 Assignment

In Verilog there are two assignment statements, blocking and non-blocking. In "net" type signal statements, these produce the same results. But, in a "reg" signal statement they produce different results.

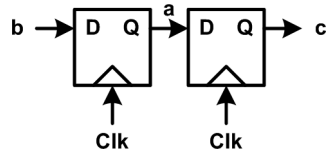
```
//Assume initial values a = 4, b = 5, c = 6
//Blocking assignment
always @(posedge clk)
```

```
begin
      a=b;   //a = b = 5; c=6;
      c=a;   //c = a = b = 5;
end
//Non-blocking assignment
always @(posedge clk)
begin
      a <= b;   //a = 5;
      c <= a;   //c = 4;
end

//Synthesis results:
//Blocking assignment
```




```
//Non-blocking assignment
```



You can easily distinguish the difference between blocking assignments and non-blocking assignments. The data transition of the blocking assignment occurs one by one and that of the non-blocking assignment occurs simultaneously. When you design a digital system, it is best to use one type of assignment only.

A.4.6 Ports and Connections

Verilog has three types of port: input, output, and inout (Figure A.8). Basically, all ports are considered as wires. Therefore, if the ports use "net" type signals, you do not need to declare wire statements. But if the output port needs to hold the value – the left side value in initial or always statements – you should declare a "reg" statement about that signal. Since the input signal cannot hold a value, you cannot declare a "reg" statement for input and inout ports.

A.4.7 Expressions

The most frequently used statements in Verilog are "initial" and "always." Both of them are used to generate events. Normally, the "always" statement is used to design sequential logic like flip–flops or to design complex combinational logic, and the

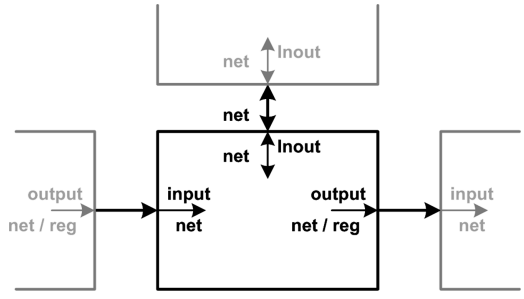


Figure A.8 Ports

"initial" statement is used to declare an initial value. One big difference between "initial" and "always" is the number of executions: the "initial" statement runs only once when the module is activated, but the "always" statement runs every time a specified event occurs. Also, the "initial" statement is omitted when the module is synthesized. Therefore, use the "initial" statement only to describe behavior or to declare an initial value.

```
//Initial statement
initial   //Execute once to initiate ing_sig and temp_wire
begin
      int_sig = 0;
      temp_data = 4'b0;
end
//Always statements
always @(posedge clk)
      //Execute every positive edge of the clock signal
begin
      int_sig = in_data [4];
      temp_data = in_data[3:0];
end
always @(a or b or c)

      //Execute whenever value of a or b or c is changed.
      //→ Sensitivity list
begin
      //Write all values used in right-hand side of the equation
      //and all values in conditions to the sensitivity list
      if (c ==1);
      d = (a & b);
end
```

As in the C language, Verilog supports "if–else" statements and "case" statements. These statements should be inside the "initial" or "always" statements. With these conditional statements there is one rule: "*Make a complete conditional statement.*" When you use "if–else" and "case" statements with edge-triggered events you do not need to make the condition complete; but when you use them with level-triggered

events, like combinational logic, you should make the condition complete. If you do not do this, undesired latches are generated in the logic, and those latches cause an unrecoverable timing violation like hold time violations and setup time violations.

```
//Edge-triggered event
always @(posedge Clk)
       //An edge-triggered event is synthesized with a flip-flop.
       //Therefore you don't need to complete the condition.
begin
       if (enable == 1) data_latch <= data_in;
end

//Combinational logic:
//if-else statement
always @(a or b or c or d or e)
       //List all right-side signals and condition signals
       //in an if-else statement.
       //There are three choices: if, else if, else.
       //Although you don't need the 'else' case,
       //you should declare it for a complete declaration.
       //If you don't complete the conditions, an undesired latch
       //will be inserted.
begin
       if (a & b == 1)         out = c;
       else if ( b & c == 0)   out = d & e;
       else                    out = 1'b0;
end

//Case statement
always @(a or b or c or d)
begin
       case({a,b})   //There are three case statements: case, casex, casez.
             2'b00 : out = 1;   //case signal can be 0 or 1
             2'b01 : out = 0;   //casex signal can be 0 or 1 or x
             default : out =0;   //casez signal can be 0 or 1 or z
             //With a case statement, declare a default case
             //to make the condition complete.
             //It prevents an unwanted latch.
       endcase
end
```

A.4.8 Instantiation

Digital systems are designed with a hierarchical structure, so that one big system consists of many sub-modules. One reason for this is design complexity. Since the system cannot be verified all at once, the desirability of dividing it into several blocks is increased. After verifying the sub-modules, it is easier to verify the big system so that the efficiency of designing is increased. Other advantages are the shortened design time and efficient use of system resources. Therefore, you often use instantiation in Verilog. There are two methods for instantiation: port assignment based on an order of signals, and port assignment based on number of ports.

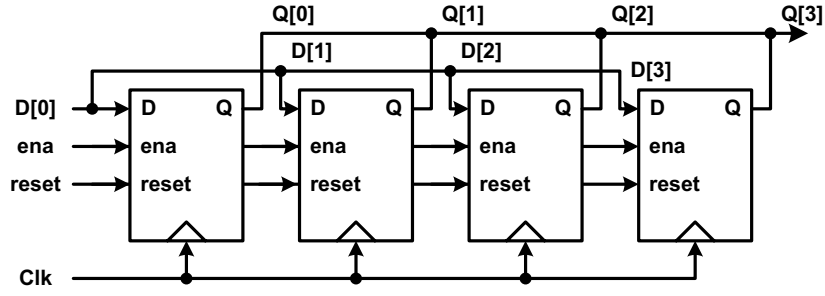


Figure A.9 Four-bit flip–flop

Examples of Instantiation: Flip–Flops

```
//1-bit flip-flop
module   DFF (Clk, reset, ena, D, Q);
      input  Clk, reset, ena;
      input  D;
      output  Q;
      always @(posedge Clk or negedge reset)
      begin
           if(~reset) Q <= 0;
           else if(ena) Q <=D;
     end
endmodule

//4-bit flip-flop (Figure A.9)
module   4b_DFF(Clk, reset, ena, D, Q);
       input  Clk, reset, ena;
       input [3:0]  D;
       output [3:0]  Q;
//Instantiation by order
       //When the sub-block has only a few in/out signals,
       //you can use instantiation by order.
       //As the signal is connected by order, it is easy to make mistakes.
       //This is especially true when debugging the block
       //or when adding or removing signals of sub-blocks.
       //Therefore, this method is not recommended for a block
       //containing lots of ports.
       DFF   DFF0(Clk, reset, ena, D[3], Q[3]);
       DFF   DFF1(Clk, reset, ena, D[2], Q[2]);
       DFF   DFF2(Clk, reset, ena, D[1], Q[1]);
       DFF   DFF3(Clk, reset, ena, D[0], Q[0]);
//Instantiation by name
       //In this method, you should declare the name or port with '.'
       //and put the name of the link signal to the port.
       //.Clk (Clk) means that the Clk signal of the 4-bit flip-flop is
       //connected to the Clk port of the 1-bit flip-flop.
       //This method may seem a little scrappy, but it does not
       //depend on the order of ports. That means this method
```

```
      //can add or remove ports of the sub-blocks, which is
      //really helpful for debugging.
      DFF  DFF0(.Clk (Clk),
           .reset (reset),
           .ena (ena),
           .D (D[3]),
           .Q (Q[3]));
     DFF  DFF1(.Clk (Clk),
           .reset (reset),
           .ena (ena),
           .D (D[2]),
           .Q (Q[2]));
    DFF  DFF2(.Clk (Clk),
           .reset (reset),
           .ena (ena),
           .D (D[1]),
           .Q (Q[1]));
    DFF  DFF3(.Clk (Clk),
           .reset (reset),
           .ena (ena),
           .D (D[0]),
           .Q (Q[0]));
endmodule
```

A.4.9 Miscellaneous

Include Directive

Like the C language's #include, Verilog supports an include directive. This is used to include other Verilog code or library file within the current Verilog code. The syntax of the include directive is shown below:

```
`include  "datapath.v"
       //Datapath.v includes adder module, multiplier, divider modules.
       //Then you can use those modules when you declare the include directive.
module
       processor(clk, inputs, output);
             input  clk;
             input  inputs;
             output  outputs;
       //Divider and multiplier are described in datapath.v.
       divider proc_div(.input(input), .output(output));
       multiplier proc_mul(.input(input), .output(output));
endmodule
```

Define Statement

The "define" statement is a kind of complier command to transpose a text. When you use a certain value recursively, the "define" statement can help you.

```
//Do not write semicolons in define statements.
//Once you have declared a define statement,you can use the text in any modules.
```

```
//You can use these defined texts in other files if you include this file.
`define    WISA_RSHA    6'b1000_00
`define    WISA_RTEX    6'b1000_01
`define    WISA_RDON    6'b1000_10
`define    WISA_TMOD    6'b0100_01
`define    WISA_MBAS    6'b0010_00
`define    WISA_RCLR    6'b0010_01
module
      decode(inputs, outputs);
            input inputs;
            output outputs;
...
     //Command decoder
     always @(ID1data)
     begin
          case(ID1data[123:118])
          //In compile time, these texts are converted into defined signals.
          `WISA_RSHA   : ID1ctrl_OP <= `WCTRL_OP_RSHA;
          `WISA_RTEX   : ID1ctrl_OP <= `WCTRL_OP_RTEX;
          `WISA_RDON   : ID1ctrl_OP <= `WCTRL_OP_RDON;
          `WISA_TMOD   : ID1ctrl_OP <= `WCTRL_OP_TMOD;
          `WISA_MBAS   : ID1ctrl_OP <= `WCTRL_OP_MBAS;
          `WISA_RCLR   : ID1ctrl_OP <= `WCTRL_OP_RCLR;
           default     : ID1ctrl_OP <= 6'b0;
           endcase
     end
...
endmodule
```

Timescale Command

In Verilog you can define a timescale. You should define this when you want to perform gate-level simulation or post-PnR simulation. Of course you can define a delay time in your design with timescale. But it will be omitted in synthesis time. Therefore use a time delay statement only for functional verification like gate propagation delay modeling.

```
`timescale   1ns / 1ps
//1ns is the reference time unit. It is a unit of time and delay.
//1ps is the precision of the time.
//Below, #5 means 5ns in simulation time,
//but this is not effective in the synthesis.
module   testbench();
      reg clk;
      initial    clk = 1b0;
      always #5
      begin
           clk = ~clk;
      end
endmodule
```

A.5 Example of Four-bit Adder with Zero Detection

See Figure A.10.

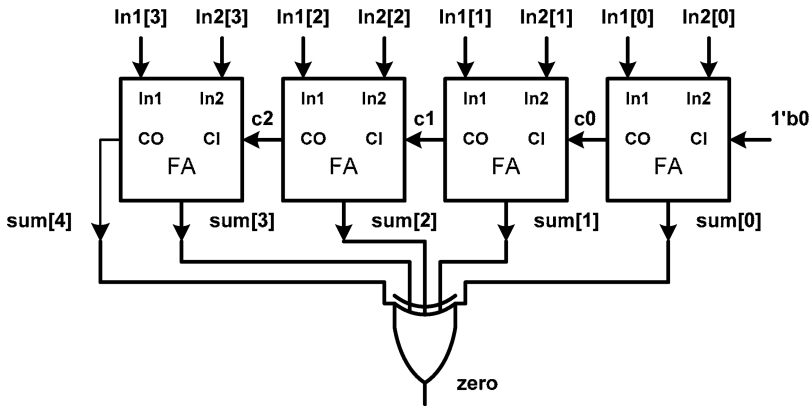


Figure A.10 Four-bit adder with zero detection

Behavior-level Description

This behavior-level description includes an initial statement and thus it is not synthesizable.

```
module    adder4 (in1, in2, sum, zero);
      input [3:0] in1, in2;
      output [4:0] sum;
      output zero;
      reg [4:0] sum;
      reg zero;
      initial
      begin
            sum = 0;
            zero = 1;
      end
      always @ (in1 or in2)
      begin
            sum = in1 + in2;
            if (sum == 0)  zero = 1;
            else  zero = 0;
      end
endmodule
```

Hierarchical Description 1 – Full Adder

```
//1-bit full adder module
module    FA (in1, in2, c_in, sum, c_out);
```

```
      input      in1, in2, c_in;
      output     sum, c_out;
      assign     {c_out, sum} = in1 + in2 + c_in;
endmodule

//4-bit full adder module
module adder4 (in1, in2, sum, zero);
      input [3:0]    in1;
      input [3:0]    in2;
      output [4:0]   sum;
      output         zero;
      wire           c0, c1, c2;
      FA add1 (. in1(in1[0]), .in2(in2[0]), .c_in(1b0), .sum(sum[0]), .c_out(c0));
      FA add2 (. in1(in1[1]), .in2(in2[1]), .c_in(c0), .sum(sum[1]), .c_out(c1));
      FA add3 (. in1(in1[2]), .in2(in2[2]), .c_in(c1), .sum(sum[2]), .c_out(c2));
      FA add4 (. in1(in1[3]), .in2(in2[3]), .c_in(c2), .sum(sum[3]),
.c_out(sum[4]));

      //zero detection
      zero = | sum;
endmodule
```

Hierarchical Description 2 – Designware

The Synopsys design compiler, a synthesis tool, supports designware libraries. A designware library is a type of datapath set – unsigned adder, subtracter, multiplier, divider, floating-point adder, subtracter, multiplier, divider, and so on. When you use designware you do not need to declare basic datapaths. More details of designware are available on the website of Synopsys: www.synopsys.com.

```
//4-bit full adder module
module    adder4 (in1, in2, sum, zero);
      input [3:0]      in1;
      input [3:0]      in2;
      output [4:0]     sum;
      output           zero;
      wire             c0, c1, c2;

      //Designware module #(bitwidth) instant_name(portlist)
      //When you declare the designware adder,
      //the type of adder is decided by timing and area bounds.
      DW01_add #(1) add0(.A(in1[0]), .B(in2[0]), .CI(1b0), .SUM(sum[0]), .CO(c0));
      DW01_add #(1) add1(.A(in1[1]), .B(in2[1]), .CI(c0), .SUM(sum[1]), .CO(c1));
      DW01_add #(1) add2(.A(in1[2]), .B(in2[2]), .CI(c1), .SUM(sum[2]), .CO(c2));
      DW01_add #(1) add3(.A(in1[3]), .B(in2[3]), .CI(c2), .SUM(sum[3]),
        .CO(sum[4]));

      //Zero detection
      zero = | sum;
endmodule
```

A.6 Synthesis Scripts

dont_use Scan Chain

This command is used to exempt some library blocks from the process library file.

```
set_dont_use    {typical/SD*}
set_dont_use    {typical/SE*}
```

read

```
read -format db {"WGR_ctrl.db"}
      //Read database as db format
read -format db {"WGR_datapath.db"}
      //This command is used to read a previously synthesized block
read -format verilog {"WGR_main.v"}
      //Used to read target design Verilog file
check_design
      //Check target design
reset_design
      //Reset target design
remove_clock -all
      //Since each block has its own clock and timing bound,
      //the previously defined clock should be removed.
```

IF: TOP

```
current_design Wif
      //Define current design.
      //One Verilog file can have several modules,
      //so you should declare the current block to be synthesized.
remove_constraint - all
      //Since the new constraints will be declared,
      //the previous constraints should be removed.
create_clock -period 40 REclk -waveform {0,20.0}
      //Create clock: period is 40ns, duty cycle is 50%.
set_dont_touch find(net, "REclk")
      //Since clock net is a global signal, declare ‘‘dont_touch’’ net.
set_input_delay 2.5 -rise -clock REclk {IFmode, SYS_nRESET}
      //Declare input signal delay.
set_load 0.05 all_outputs()
      //Set load to all output signals
      //to adjust driving power of the output load
set_load 0.3 InREQ
      //Set load to output signal.
set_max_transition 0.06 all_outputs()
      //To limit rising or falling time due to RC time constant.
set_max_area 39
      //Declare the area bound.
compile -map_effort high
      //Set compile option.
report_area > Wif_Area.txt
      //Write area report.
```

```
report_timing -from all_inputs() -to all_outputs() > Wif_Timing.txt
      //Write timing report.
      //Timing report of the synthesis only includes gate propagation delay.
write -format db -hierarchy -output "Wif.db"
      //Write result as db format – for hierarchical synthesis
write -format verilog -hierarchy -output "Wif.v"
      //Write result as Verilog file – for gate-level simulation.
write_sdf "/home/denber/project/ramp-GR/verilog/sdf/Wif.sdf"
      //Write SDF file – standard delay file.
```

Script for Top Module: WRE_top

```
current_design WRE
      //Declare current design.
remove_clock – all
      //Remove previously defined clock.
remove_constraint – all
      //Remove previously define constraints.
create_clock "REclk" -period 40.0 -waveform {0, 20.0}
      //Create clock for processor.
create_clock "memclk1" -period 20.0 -waveform {0, 10.0}
      //Create clock for memory module.
set_dont_touch find (net, "REclk")
      //Declare dont_touch net signal – clock.
set_dont_touch find (net, "memclk1")
      //Declare dont_touch net signal – clock.
set_dont_touch find (cell, "GR_IF")
      //Declare previously synthesized module.
      //In top module, it only uses the previously synthesized module.
set_dont_touch find (cell, "GR_ID1")
set_dont_touch find (cell, "GR_ID2")
set_load 0.05 {Dbg_out, CSR}
      //Set output load capacity.
set_max_transition 0.06 {Dbg_out, CSR}
      //Limit rising and falling times.
compile -map_effort high
      //Set compiler option.
report_area > WRE_Area.txt
      //Write area report.
report_timing -from all_inputs() -to all_outputs() > WRE_Timing.txt
      //Write timing report.
write -format db -hierarchy -output "WGR.db"
      //Write result as db format
write -format verilog -hierarchy -output "WGR.v"
      //Write result as Verilog file.
write_sdf "WGR.sdf"
      //Write SDF file.
```